Lazy Data-Oriented Evaluation Strategies
3rd ACM SIGPLAN Workshop on Functional High-Performance Computing

Prabhat Totoo and Hans-Wolfgang Loidl

Heriot-Watt University, Edinburgh

Sep 4, 2014
Gothenburg, Sweden
The talk is about using **laziness** to make *parallel programs run faster*.
1 Intro and Motivation
2 Parallel Haskell - GpH
3 Examples: Primitives and Evaluation Strategies
4 Tree Strategies
   1 Basic Strategies and Parallelism Control
   2 Advanced Strategies and Parallelism Control
5 Performance Evaluation
6 Summary
7 Ongoing Work
Intro and Motivation

- What we want to achieve:
  - higher performance through more flexible parallelism control
- How:
  - through the use of lazy evaluation and circular programming techniques
  - develop a number of advanced parallelism control mechanisms
  - embed them into evaluation strategies
- Performance results:
  - comparative study of performance using a constructed test program and a Barnes-Hut algorithm
Glasgow parallel Haskell (GpH)

- support for semi-explicit parallelism through GpH extension
- GpH Primitives
  - `par` to specify parallelism
    \[ x \ 'par' \ y \Rightarrow y \]
    \(x\) is sparked to be potentially evaluated in parallel.
  - `pseq` to enforce sequential ordering
    \[ x \ 'pseq' \ y \Rightarrow y \]
    \(x\) is evaluated to WHNF.
  - purely functional, stateless code

Evaluation Strategies

- build on top of basic primitives
- raise the level of abstraction even higher
- separate coordination from computation aspects

```haskell
data Eval a = Done a
runEval :: Eval a -> a
runEval (Done x) = x

type Strategy a = a -> Eval a
rseq, rpar :: Strategy a
rseq x = x 'pseq' Done x
rpar x = x 'par' Done x

using :: a -> Strategy a -> a
x 'using' strat = runEval (strat x)
```
Examples

sequential factorial

```
-- factorial example

fact m n
  | m == n = m
  | otherwise =
        (left * right)

where
  mid   = (m + n) \text{\texttt{div}} 2
  left  = fact m mid
  right = fact (mid + 1) n
```
introducing parallelism using primitives

```haskell
-- factorial example
fact m n
  | m == n = m
  | otherwise = left `par` right `pseq`
    (left * right)
  where
    mid = (m + n) `div` 2
    left = fact m mid
    right = fact (mid + 1) n
```
GpH Primitives and Evaluation Strategies

Examples

sequential factorial

```haskell
-- factorial example
fact m n
| m == n = m
| otherwise = (left * right)

where
    mid  = (m + n) `div` 2
    left = fact m mid
    right = fact (mid + 1) n
```
GpH Primitives and Evaluation Strategies

Examples

using evaluation strategies

```haskell
-- factorial example
fact m n
  | m == n = m
  | otherwise =
    (left * right)

where
  mid   = (m + n) 'div' 2
  left  = fact m mid
  right = fact (mid + 1) n
strategy result = do
  rpar left
  rseq right
  return result
```

define strategy separate from algorithm
using evaluation strategies

```
-- factorial example
fact m n
| m == n = m
| otherwise =
  (left * right) 'using' strategy
  where
    mid = (m + n) 'div' 2
    left = fact m mid
    right = fact (mid + 1) n
strategy result = do
  rpar left
  rseq right
  return result
```

apply strategy with using
GpH Primitives and Evaluation Strategies (2)

Examples

Primitives

```haskell
-- factorial example
fact m n
  | m == n = m
  | otherwise = left `par` right `pseq`
    (left * right)
  where
    mid = (m + n) `div` 2
    left = fact m mid
    right = fact (mid + 1) n
```

Evaluation Strategies

```haskell
-- factorial example
fact m n
  | m == n = m
  | otherwise = (left * right)
    `using` strategy
  where
    mid = (m + n) `div` 2
    left = fact m mid
    right = fact (mid + 1) n
    strategy result = do
      rpar left
      rseq right
      return result
```

– clear separation of coordination from computation code
– more structured parallel program
GpH Primitives and Evaluation Strategies (2)

Examples

Primitives

```haskell
-- factorial example
fact m n
| m == n = m
| otherwise = left 'par' right 'pseq'
  (left * right)
where
  mid = (m + n) 'div' 2
  left = fact m mid
  right = fact (mid + 1) n
```

Evaluation Strategies

```haskell
-- factorial example
fact m n
| m == n = m
| otherwise = (left * right)
  'using' strategy
where
  mid = (m + n) 'div' 2
  left = fact m mid
  right = fact (mid + 1) n

strategy result = do
  rpar left
  rseq right
  return result
```

- clear separation of coordination from computation code
- more structured parallel program

Data parallel strategies

```haskell
-- e.g. parallel map
parMap strat f xs =
  map f xs 'using' parList strat
-- where strat specifies the eval degree
```

```haskell
[_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]
```

Totoo and Loidl (Heriot-Watt Uni) FHPC’14 Gothenburg
GpH Primitives and Evaluation Strategies (2)

Examples

Primitives

```
-- factorial example
fact m n
| m == n = m
| otherwise = left 'par' right 'pseq'
   (left * right)
where
  mid = (m + n) 'div' 2
  left = fact m mid
  right = fact (mid + 1) n
```

Evaluation Strategies

```
-- factorial example
fact m n
| m == n = m
| otherwise = (left * right)
where
  mid = (m + n) 'div' 2
  left = fact m mid
  right = fact (mid + 1) n
strategy result = do
  rpar left
  rseq right
  return result
```

Data parallel strategies

```
-- e.g. parallel map
parMap strat f xs =
  map f xs 'using' parList strat
-- where strat specifies the eval degree

-- chunking to control granularity
parMapChunk strat f xs =
  map f xs 'using' parListChunk size strat
```

- clear separation of coordination from computation code
- more structured parallel program
Tree Strategies
Tree Strategies

- 2 classes of strategies
  1. Basic Strategies
     - more general
     - use no/traditional parallelism control mechanisms
     - e.g. parTree, parTreeDepth
  2. Advanced Strategies
     - use advanced mechanisms
       - flexible
       - use laziness inherently
       - fuel-based control
     - e.g. parTreeLazySize, parTreeFuelXXX
Basic Strategies

Naive with no parallelism control

= node
Basic Strategies
Naive with no parallelism control

- **parTree**
  - analogous to **parList**
  - uncontrolled spark creation – high overhead!
  - basic implementation of traverse from Traversable typeclass
Traditional Parallelism Control Mechanisms

Depth-thresholding with \texttt{parTreeDepth}

\begin{center}
\begin{tikzpicture}
  \node (root) at (0,0) {$d_0$};
  \foreach \i in {1,...,4}
  \node (l\i) at (\i,-2) {};\node (r\i) at (\i,-2) {};
  \foreach \i in {1,...,2}
  \node (l2\i) at (\i,-4) {};\node (r2\i) at (\i,-4) {};
  \node (l31) at (-1,-6) {};\node (l32) at (-1,-6) {};\node (r31) at (-1,-6) {};\node (r32) at (-1,-6) {};
  \node (l41) at (-2,-8) {};\node (l42) at (-2,-8) {};\node (r41) at (-2,-8) {};\node (r42) at (-2,-8) {};
  \foreach \i in {1,...,4}
  \draw (l\i) -- (l\i);\draw (r\i) -- (r\i);\draw (l\i) -- (l\i);\draw (r\i) -- (r\i);\draw (l2\i) -- (l2\i);\draw (r2\i) -- (r2\i);\draw (l31) -- (l31);\draw (l32) -- (l32);\draw (l41) -- (l41);\draw (l42) -- (l42);\draw (r31) -- (r31);\draw (r32) -- (r32);\draw (r41) -- (r41);\draw (r42) -- (r42);
\end{tikzpicture}
\end{center}

\begin{itemize}
\item simple, low overhead and predictable parallelism
\item works pretty well for regular tree
\item lacks flexibility for unbalanced trees – parallelism may not reside in the top level
\end{itemize}
Traditional Parallelism Control Mechanisms

Depth-thresholding with parTreeDepth

Summary:
▶ simple, low overhead and predictable parallelism
▶ works pretty well for regular tree
▶ lacks flexibility for unbalanced trees – parallelism may not reside in the top level

Info flow down
Context path length
Parameter $d$
Traditional Parallelism Control Mechanisms

Depth-thresholding with \texttt{parTreeDepth}

\begin{itemize}
  \item simple, low overhead and predictable parallelism
  \item works pretty well for regular tree
  \item lacks flexibility for unbalanced trees – parallelism may not reside in the top level
\end{itemize}
Traditional Parallelism Control Mechanisms

Depth-thresholding with parTreeDepth

- **Summary:**
  - simple, low overhead and predictable parallelism
  - works pretty well for regular tree
Traditional Parallelism Control Mechanisms

Depth-thresholding with \texttt{parTreeDepth}

\begin{itemize}
  \item \textbf{Summary:}
  \begin{itemize}
    \item \textbf{simple, low overhead} and predictable parallelism
    \item works pretty well for regular tree
    \item lacks flexibility for unbalanced trees – parallelism may not reside in the top \( d \) level
  \end{itemize}
\end{itemize}
Traditional Parallelism Control Mechanisms

Synthesised size info as threshold

Info flow up
Context global
Parameter s
Traditional Parallelism Control Mechanisms

Synthesised size info as threshold

- size synthesised in a single annotation pass
Traditional Parallelism Control Mechanisms

Synthesised size info as threshold

- size synthesised in a single annotation pass
- ensures sparks are not created for smaller sub-trees e.g. $s < 20$
Traditional Parallelism Control Mechanisms

Synthesised size info as threshold

- Size synthesised in a single annotation pass
- Ensures sparks are not created for smaller sub-trees e.g. $s < 20$
- Carries administrative overhead
Lazy size computation

- removes the need for initial annotation traversal
- lazily checks size of subnodes evaluating only up to what is needed
- size check function is implemented using (algebraic) natural instead of (atomic) integer type

\[ T = \text{uneval thunk} \]

```
-- returns tree when it has established that the sub-tree contains at least s nodes without a full deconstruction.
isBoundedSize s t = lazy_check t > s

lazy_check :: QTree tl tn -> Natural
lazy_check = ...
```
Advanced Mechanisms

Fuel-based control

- fuel
  - limited resources distributed among nodes
  - similar to “potential” in amortised cost
  - and the concept of “engines” to control computation in Scheme

- parallelism generation (sparks) created until fuel runs out

- more flexible to throttle parallelism
Advanced Mechanisms

Fuel-based control

- fuel split function
  - flexibility of defining custom function specifying how fuel is distributed among sub-nodes
  - e.g. pure, lookahead, perfectsplit
  - split function influences which path in the tree will benefit most of parallel evaluation

annotate tree with fuel info based on split_func
Fuel-based Control Mechanism

- pure
- lookahead
- perfectsplit

Diagram: A tree structure with nodes labeled from $s_0$ to $s_{11}$, representing states or nodes in the control mechanism.

Parameters:
- $f$: fuel
- $f/2$: half of fuel
- $f_0$, $f_1$, $f_2$: specific fuel levels

Flow:
- Info flow down
- Context local
- Parameter

Flow limited:
- Info flow down/limited
- Context local ($N$)
- Parameter

Totoo and Loidl (Heriot-Watt Uni)

FHPC’14 Gothenburg
Fuel-based Control Mechanism

pure, lookahead, perfectsplit

- Characteristics of **pure** version
  - splits fuel equally among sub-nodes
Fuel-based Control Mechanism

pure, lookahead, perfectsplit

- Characteristics of pure version
  - splits fuel equally among sub-nodes
  - fuel lost on outer nodes
Fuel-based Control Mechanism

pure, lookahead, perfectsplit

Characteristics of lookahead version

- looks ahead $N$ level down before distributing unneeded fuel
- more efficient distribution
Fuel-based Control Mechanism

pure, lookahead, perfectsplit

Characteristics of perfectsplit version
- perfect fuel splitting
- distributes fuel based on sub-node sizes
Advanced Mechanisms

Fuel-based control

- bi-directional fuel transfer – *giveback* version
  - fuel is passed down from root
  - fuel is given back if tree is empty or fuel is unused
  - *giveback* mechanism is implemented via *circularity*

- fuel represented using list of values instead of an (atomic) integer
- *giveback* mechanism is effective in enabling additional parallelism for irregular tree
  - distribution carries deeper inside the tree
Fuel-based Control Mechanism

giveback fuel flow

<table>
<thead>
<tr>
<th>giveback</th>
<th>Info flow</th>
<th>down/up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>local</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>$f$</td>
<td></td>
</tr>
</tbody>
</table>
Fuel-based Control Mechanism

giveback fuel flow

\( f_{in} \): fuel down

\( f_{out} \): fuel up

\( f' \): fuel reallocated

giveback

- Info flow: down/up
- Context: local
- Parameter: \( f \)
Fuel-based Control Mechanism

giveback fuel flow

- f_in: fuel down
- f_out: fuel up
Fuel-based Control Mechanism

giveback fuel flow

- \( f_{\text{in}} \): fuel down
- \( f_{\text{out}} \): fuel up
- \( f_{\text{in}'} \): fuel reallocated
Advanced Mechanisms
Fuel-based control with giveback using circularity

```haskell
-- | Fuel with giveback annotation
annFuel_giveback :: Fuel -> QTree tl -> AnnQTree Fuel tl
annFuel_giveback f t = fst $ ann (fuelL f) t
    where
      ann :: FuelL -> QTree tl -> (AnnQTree Fuel tl, FuelL)
      ann f_in E = (E, f_in)
      ann f_in (L x) = (L x, f_in)
      ann f_in (N (Q a b c d)) = (N (AQ (A (length f_in) a’ b’ c’ d’)),
                                 emptyFuelL)
      where
        (f1_in:f2_in:f3_in:f4_in:_)) = fuelsplit numnodes f_in
        (a’, f1_out) = ann (f1_in ++ f4_out) a
        (b’, f2_out) = ann (f2_in ++ f1_out) b
        (c’, f3_out) = ann (f3_in ++ f2_out) c
        (d’, f4_out) = ann (f4_in ++ f3_out) d
```
Advanced Mechanisms
Fuel-based control with giveback using circularity

```haskell
-- | Fuel with giveback annotation
annFuel_giveback :: Fuel -> QTree tl -> AnnQTree Fuel tl
annFuel_giveback f t = fst \$ ann (fuelL f) t
  where
    ann :: FuelL -> QTree tl -> (AnnQTree Fuel tl, FuelL)
    ann f_in E = (E, f_in)
    ann f_in (L x) = (L x, f_in)
    ann f_in (N (Q a b c d)) = (N (AQ (A (length f_in)) a' b' c' d'),
                              emptyFuelL)
      where
        (f1_in:f2_in:f3_in:f4_in:_ = fuelsplit numnodes f_in
        (a', f1_out) = ann (f1_in ++ f4_out) a
        (b', f2_out) = ann (f2_in ++ f1_out) b
        (c', f3_out) = ann (f3_in ++ f2_out) c
        (d', f4_out) = ann (f4_in ++ f3_out) d
```

Totoo and Loidl (Heriot-Watt Uni)
FHPC'14 Gothenburg
18 / 25
Advanced Mechanisms

Fuel-based control with giveback using circularity

-- | Fuel with giveback annotation

`annFuel_giveback::Fuel -> QTree tl -> AnnQTree Fuel tl`

`annFuel_giveback f t = fst \$ ann (fuelL f) t`

```
where

ann::FuelL -> QTree tl -> (AnnQTree Fuel tl,FuelL)

ann f_in E = (E,f_in)
ann f_in (L x) = (L x,f_in)
ann f_in (N (Q a b c d)) = (N (AQ (A (length f_in)) a’ b’ c’ d’), emptyFuelL)

where

(f1_in:f2_in:f3_in:f4_in:_ ) = fuelsplit numnodes f_in
(a’,f1_out) = ann (f1_in ++ f4_out) a
(b’,f2_out) = ann (f2_in ++ f1_out) b
(c’,f3_out) = ann (f3_in ++ f2_out) c
(d’,f4_out) = ann (f4_in ++ f3_out) d
```

- fuel flows back in a circular way
## Tree Strategies

### Summary

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Type</th>
<th>Info flow</th>
<th>Context</th>
<th>Parameter</th>
<th>Heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>parTree</td>
<td>element-wise sparks</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>parTreeDepth</td>
<td>depth threshold</td>
<td>down</td>
<td>path length</td>
<td>$d$</td>
<td>yes</td>
</tr>
<tr>
<td>parTreeSizeAnn</td>
<td>annotation-based</td>
<td>up</td>
<td>global</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>parTreeLazySize</td>
<td>lazy size check</td>
<td>down</td>
<td>local</td>
<td>$s$</td>
<td>yes</td>
</tr>
<tr>
<td>parTreeFuelAnn</td>
<td>annotation-based</td>
<td>down</td>
<td>local</td>
<td>$f$</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>- pure</td>
<td>down</td>
<td>local</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- lookahead</td>
<td>down/limited</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- giveback</td>
<td>up/down</td>
<td>local</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- perfectsplit</td>
<td>down</td>
<td>global</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance Evaluation

Setup

- **Machine**
  - 48-core *server-class many-core* (1.4Ghz)
  - 8 NUMA regions (remote region access is 2.2x local access)
  - 64GB RAM
  - running Linux

- **Compiler:** ghc-7.6.1

- **Libraries:**
  - parallel-3.2
  - pardata-0.1 extended set of advanced strategies for tree-like data structures (includes heuristics for auto-tuning)

- **Applications:** test program, Barnes-Hut, sparse matrix multiplication, (LSS)
Performance Evaluation (1)
Test program speedups on 1-48 cores. 100k elements.

- normal depth distr.
- homo/hetero comp.

- performance: improvement of > 18% (depth vs. best fuel)
- giveback hitrate e.g. for $f = 100$, number of hits=478
Performance Evaluation (2)

Barnes-Hut speedups on 1-48 cores. 2 million bodies. 1 iteration.

- pure fuel gives best perf. – simple but cheap fuel distr.; lookahead/giveback within 6/20%
- fuel ann/unann overheads: 11/4% for 2m bodies
- more instances of giveback due to highly irregular input (7682 for 100k bodies, \( f = 2000 \))
Summary

- we use laziness to improve parallel performance
- we use laziness and circular programs in the coordination code to achieve additional flexibility
- we develop a number of flexible parallelism control mechanisms in the form of evaluation strategies
- we demonstrate improved performance on a constructed program and 2 non-trivial applications, in particular, with irregular trees
Ongoing Work

Graph Strategies

- develop similar evaluation strategies for graphs
- depth-first and breadth-first traversal strategies on graphs
- apply techniques (e.g. thresholding, fuel) to graph strategies
- algorithms: shortest path, max clique

SICSA Multicore Challenge III¹

¹http://www.macs.hw.ac.uk/sicsawiki/index.php/Challenge_PhaseIII
Paper and sources

- Full paper, sources for strategies and test programs:
  http://www.macs.hw.ac.uk/~dsg/gph/papers/abstracts/fhpc14.html
- Email: \{pt114, h.w.loidl\}@hw.ac.uk

Thank you!